THE OFFICE OF THE STATE CHIEF INFORMATION OFFICER
ENTERPRISE TECHNOLOGY STRATEGIES

North Carolina Statewide Technical Architecture

Domain White Paper

Application Architecture Technology Overview

STATEWIDE TECHNICAL ARCHITECTURE

Domain White Paper: Application Architecture Technology Overview

Initial Release Date:	August 1, 2003	Version:	1.0.0
Revision Approved Date:	Not Applicable		
Date of Last Review:	March 11, 2004	Version:	1.0.1
Date Retired:			
Architecture Interdependencies:			

Reviewer Notes: This is a move without modification from the old format to the new format as provided herein. A subsequent review and update will occur at a future, as yet to be determined, date. The ComponentWare section was originally approved 5/6/1997 but went through an update 9/5/2000. The Accessibility section was included and approved 6/6/2002. Published August 1, 2003.

Reviewed and updated office title and copyright date. Added a hyperlink for the ETS email – March 11, 2004.

Mission Statement

Application Architecture identifies criteria and techniques associated with the design of applications that can be easily modified to respond quickly to the state's changing business needs.

The State of North Carolina, like many private enterprises, relies heavily on computer applications to support its business operations. Because the state's business processes change dynamically in response to legislation and demands from citizens, it is important that computer applications also be able to change rapidly. Applications capable of being easily and quickly modified are called "adaptive systems."

To date, most applications developed by the state are either large and monolithic or two-tier client/server applications. The existing application inventory reflects not only the tools available at the time the applications were developed, but also how system development projects were funded. Applications were originally designed and funded to perform a specific operation, for a specific agency, and were developed independently using different languages and tools (See Figure 1). The ability to communicate with other applications was not an original design requirement, however, as legislative requirements changed, inter-application communication was required and custom interfaces were built to address this need.

The existing application architecture adversely impacts the state's business in three ways:

- The cost and time associated with modifying existing applications to support new business requirements.
- The difficulty in integrating applications to share common services and data.
- The expense of developing, using, and maintaining new applications because there is little reuse of code between software.

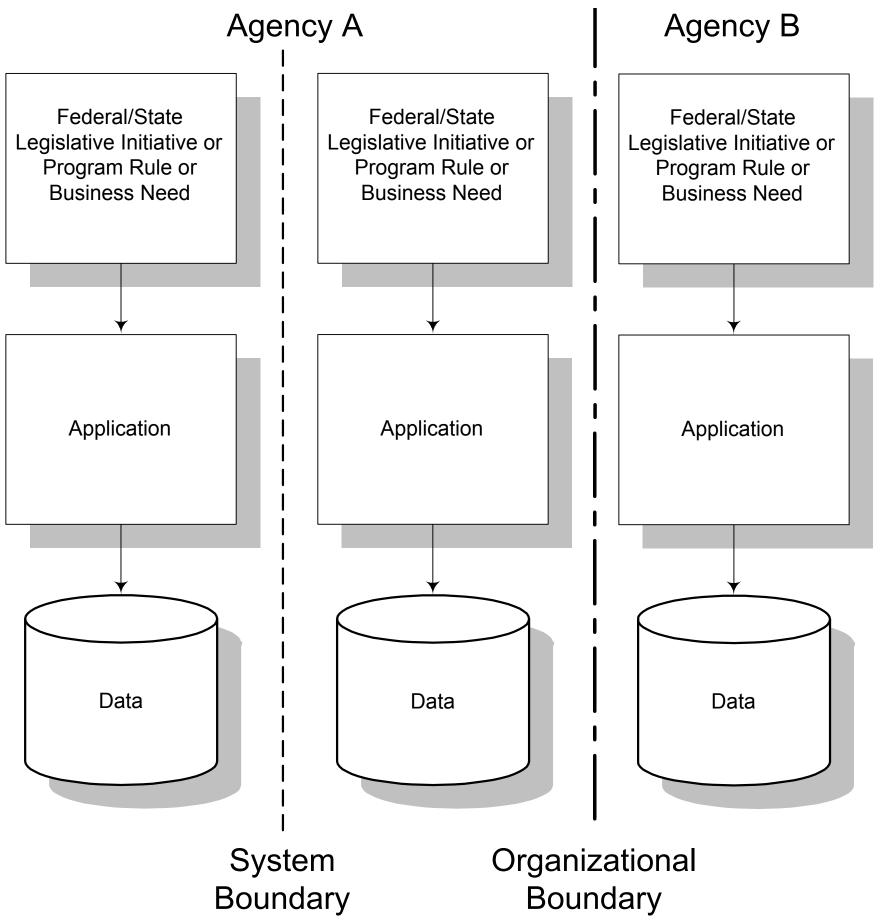
Agency A
Agency B
Federal/State Legislative Initiative or Program Rule or Business Need
Federal/State Legislative Initiative or Program Rule or Business Need
Federal/State Legislative Initiative or Program Rule or Business Need
Application
Application
Application
Data
Data
Data
System Boundary
Organizational Boundary

Figure 1. Monolithic Applications were developed and operated independently.

Recently, application development tools and technology have evolved to address these problems. Unlimited options exist for meeting business needs and delivering information to people when and where they need it.

- Units of code previously duplicated in many applications can be packaged into components or services and reused by different applications.
- Middleware allows applications to communicate with each other, access data residing on different platforms, and access the shared services.
- New user interface devices, such as web browsers, pagers, and voice response units (VRUs), have been introduced.

Implementing these components in an *N*-tier, client/server application architecture creates solutions to satisfy the state's ever-changing business needs.
Here are just a few examples that address issues facing the state today:

- If the code that exists in many applications to calculate an individual's age and perform other routine date-related functions were packaged into services shared by many applications, the effort required to make the state's application portfolio century-compliant would be far less formidable than the job now facing the state.

- Using middleware, an application issuing fishing licenses could easily access an application verifying child support payments, even if the applications are developed and maintained by different agencies.
- When an inmate convicted of certain offenses is released from prison, an application supporting the prison release process could automatically issue notices to local authorities and victims. In this way, the application is proactive; it can "push" information to users rather than requiring users to "pull" information out of databases when necessary.

There are several ways applications can be designed to maximize business flexibility.

- ***Logical application boundaries***. Applications should be designed along logical application boundaries to mimic the business processes they support. For example, suppose the state decided to establish regional textbook warehouses to supply books to the public schools. When an order is placed from the textbook requisition application, an order message is sent across the logical application boundary to the textbook inventory application. The inventory application would then decide which warehouse should ship the books, based on proximity to the requester and availability of the requested textbooks. The requisition system would not even know there are multiple warehouses. This design allows changes to warehouse locations (e.g., the addition of a new warehouse or the consolidation of warehouses) to occur without disruption to the requisition process. With this design, the application gathering data and sending the requisition "trusts" the inventory application to route the order to the most appropriate warehouse. (See Figure 2)
- ***Asynchronous processing***. Applications can be designed to take advantage of new methods of communication, such as asynchronous processing. Just as voice mail permits communication without requiring both parties to be available at the same time, asynchronous messaging technologies permit the same "de-coupling" of applications. Rather than designing an application to batch requests and send the requests to a second application, the requests can be placed in a queue and the second application has the flexibility to process the requests when it is ready (e.g., once a day, once an hour, or as they occur).

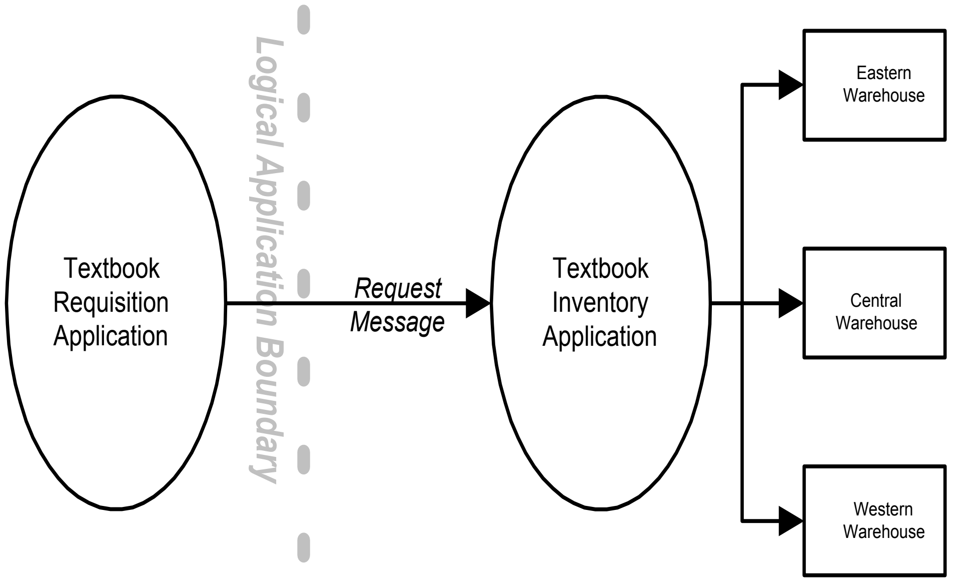
Textbook Requisition Application
Logical Application Boundary
Request Message
Textbook Inventory Application
Eastern Warehouse
Central Warehouse
Western Warehouse

Figure 2. Logical Application Boundary

- ***Using components***. Designers can build flexibility, scalability, and extensibility into applications by using components as application building blocks, much as autoworkers assemble cars on a production line. Combining pre-built and pre-tested components with new components accelerates the design, development, and delivery of new applications. With this approach, each component addresses a single business rule. If the business rule changes, only one component must be modified.

Greatest efficiency is achieved by combining new technology (e.g., middleware and components), applications designed for flexibility, and methodologies fostering a culture of reuse. But to do this, the developer's role must change. Changing business needs do not allow time for artistry or craftsmanship. New roles will evolve within the state's IT organizations. Technicians having specialized skills are needed to:

- Identify, analyze, and understand discrete business processes.
- Design, develop, test, and maintain components.
- Architect applications.
- Optimize inter- and intra-application communications.

Additional recommendations for transitioning to and maximizing the benefits of a client/server application architecture are discussed in the Recommended Best

Practices later in this chapter. Also, refer to the Middleware Architecture and Componentware Architecture chapters for more information about important aspects of the Statewide Technical Architecture.

Technical Comparisons of Application Architectures

This section discusses application design approaches in a historical sense. It shows options for implementing a client/server application architecture by comparing monolithic, two-tier, three-tier, *N*-tier, and service-oriented applications.
All computer applications -- regardless of what they do and the technology with which they are implemented -- have three general areas of functionality:

- *Business rules*: Business rules are the parts of the business process that computer applications automate.
- *Data access*: Data access code automates the storing, searching, and retrieving of data by computer applications.
- *Interface*: The interface allows applications to communicate with applications and people.

The ways in which these application functions are assembled determines:

- The flexibility of the applications.
- How quickly they can be modified to support changes in business and technology.
- How easily they interface with people and with each other.

Monolithic applications

Monolithic applications are applications where the code that implements the business rules, data access, and user interface are all tightly coupled together as part of a single, large computer program. A monolithic application must be deployed on a single platform, usually a mainframe or midrange machine. (See Figure 3)

Monolithic computer applications are deployed across the state. Since the state has many programs providing services to its citizens, it has many computer applications supporting those programs. These applications were developed independent of each other using different combinations of technology. For example, one agency application may use COBOL, CICS, and VSAM, while another application supporting the same group of citizens is implemented using COBOL and IMS.

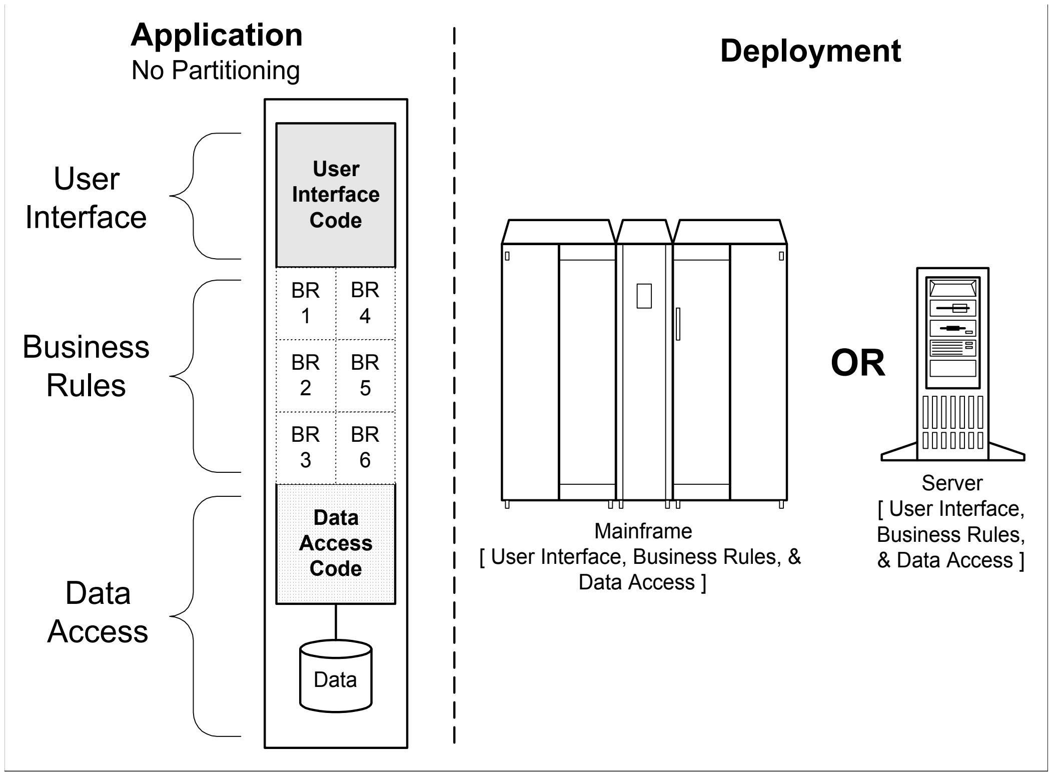

Figure 3. A monolithic application

Monolithic applications have the following drawbacks:

1. *It is costly and time consuming to modify them.* Changing the code that implements a business rule risks impacting other code in the application. When any code in the application changes, the entire application must be retested and redeployed.
2. *It is difficult to integrate applications to share services and data.* Most monolithic applications do not have well-defined interfaces that can be accessed by other applications.
3. *There is little reuse of redundant code between applications, making it more expensive to build and maintain applications.* Many applications contain functionality already replicated in other applications. Applications are slower and more costly to build, because existing functionality is reinvented many times. Applications are more expensive to operate, since the same data must be gathered, entered, and stored in many places.
4. *It is difficult to have applications communicate with other applications.* Most existing applications do not have the ability to communicate with other applications within an agency and with applications in other agencies.

5. *Monolithic applications can be accessed using only a single user interface*. Most can only be accessed via 3270 terminals. Having a single user interface is a limitation when application services need to be accessed from other user interfaces such as web browsers or the telephone (via VRUs).
6. *There is no flexibility in where the applications can be deployed*. Applications must be deployed on a single machine, usually a mainframe, to get enough processing capacity to process all parts of the application: the user interface, the business rules, and the data access code.

Two-tier client/server applications

Like many organizations, some state agencies attempted to overcome the business impact of monolithic applications by adopting client/server technology for new applications. The terms "client/server", "client", and "server" are often misunderstood. Many believe that "client/server" means an application with a graphical user interface and a relational database (neither is necessarily true). In fact, client/server applications are constructed of software "clients" that, in order to perform their required function, must request assistance -- "service" -- from other software components, known as "servers." Middleware provides communication between the client and server. Refer to the Middleware Architecture chapter for more information about how communication middleware is used in client/server applications.

These early client/server applications used architecture dictated by the tools employed in their construction. As a result, most of these applications use a *two-tier client/server architecture*. The "tiers" of client/server applications refer to the number of executable components into which the application is partitioned, not to the number of platforms where the executables are deployed. Sometimes the tiers into which the application is partitioned is called "logical partitioning", and the number of physical platforms on which it is deployed is called "physical partitioning."

In a two-tier client/server architecture, application functionality is partitioned into two executable parts, or "tiers." One tier contains both the code that implements a graphical user interface (GUI) and the code that implements the business rules. This tier executes on PCs or workstations and requests data from the second application tier, which usually executes on the machine where the application's data is stored.

This model is referred to as *two-tier, fat client*, because the application is partitioned into two tiers of executable code, and most of the application's code is contained in the tier executing on the workstations, known as the "fat client." (See Figure 4) Since business rules are tightly integrated with user interface code, the code implementing the business rules must be deployed on the same platform(s) as the user interface, and the entire workstation-resident portion of the application must be redeployed when either a business rule or the user interface changes. When the number of workstations used is high or geographically dispersed, the maintenance costs for two-tier, fat client applications escalate quickly.

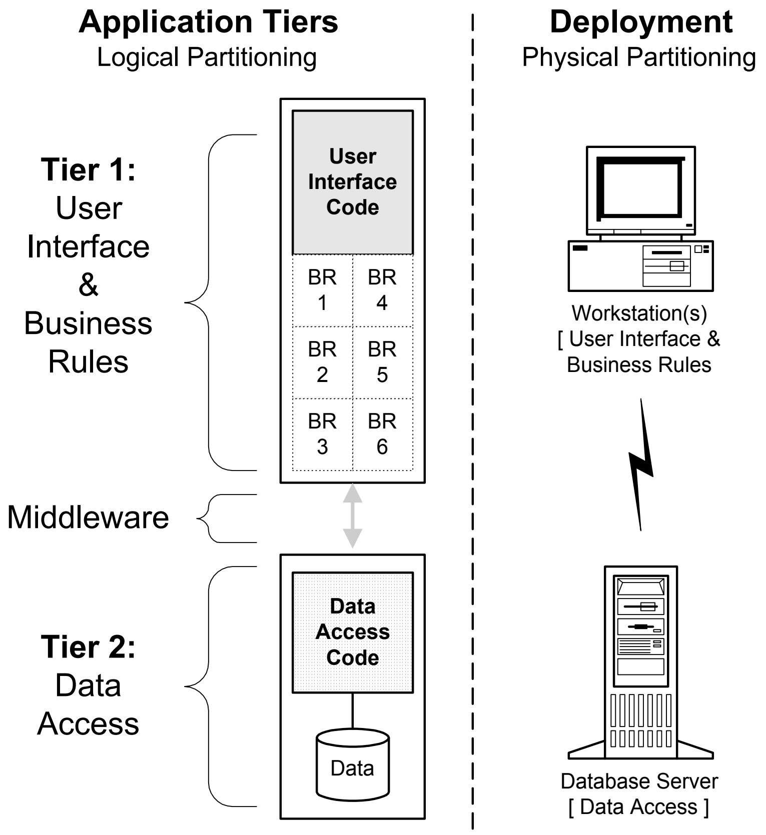

Figure 4. A two-tier, fat client application

Other client/server applications are partitioned into two tiers, but much of the code that implements the business rules is tightly integrated with the data access code, sometimes in the form of database stored procedures and triggers. This model is called *two-tier, fat server*. (See Figure 5) Two-tier, fat server applications are often implemented as mainframe applications that have web browsers for the user interfaces. This approach is actually a good first step in migrating to a three-tier or *N*-tier application architecture. Users can enjoy the ease-of-use provided by the web's graphical interface while developers work to update other parts of the application.

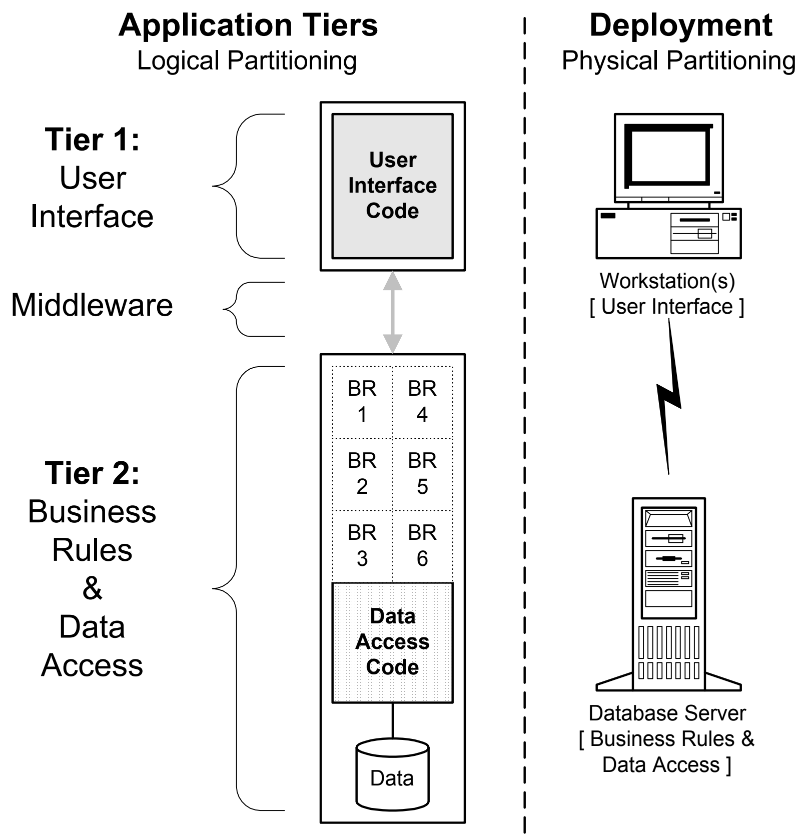

Figure 5. A two-tier, fat server application

Since the business rules in two-tier applications are tightly integrated with either the user interface code or the data access code, two-tier client/server applications have the following drawbacks:

1. *They are difficult and expensive to modify when business requirements change.* Business rules are mostly monolithic. Changing any business rule impacts the rest of the application.
2. *There is little reuse of redundant code.* It is difficult to reuse any business rules elsewhere (e.g., in other computer applications that require similar services or in batch processing that is part of the same application).
3. *There is little flexibility in selecting the platforms where the application will be deployed.* In two-tier, fat client applications, the business rules *must* execute on the same platform as the user interface, because the code they are implemented in is tightly coupled with the interface. Likewise, in two-tier, fat server applications, the business rules can only execute on the machine that hosts

the database, because they are implemented either with the database or inside the database.

4. *They can only be accessed by users with PCs running a graphical user interface.* Since the user interface is graphical, and requires a workstation to run, users with other I/O devices are excluded from using the application. These devices include existing non-graphics terminals (e.g., UNIX terminals or 3270 terminals), telephone interfaces via VRUs, and web browsers.
5. *They are more difficult to manage than monolithic applications.* Any change to either business rules or GUI means that the entire workstation-resident portion of the application must be redistributed and installed on every workstation that uses the application. Frequent software distribution can be time-consuming and logistically difficult to manage.

Three-tier client/server applications

Some client/server applications are partitioned into three executable tiers of code: user interface, business rules, and data access. Three-tier client/server means that the application's code is partitioned into three tiers. It does not imply that the three tiers execute on three different platforms. Often, the business rule tier is deployed on the same platform as the data access tier; or on the same platform(s) as the user interface. There is more flexibility in where application executables can be deployed, and may be considered a good transition step from monolithic or two-tier applications.

Three-tier client/server applications still suffer from some of the limitations of two-tier and monolithic applications. Since the business rules are monolithic:

- Changes to any business rule require re-linking, retesting, and redeploying the entire executable containing all business rules.
- There is no flexibility in where any given business rule can be deployed, since all business rules are tightly coupled in the monolithic tier and, therefore, must be deployed on the same platform.

Figure 6 illustrates a three-tier client/server application. Notice that in the deployment, or physical partitioning, of the application the business rules are separate from both the user interface and the data access code. Business rules are deployed on their own server or on the same server as the database. Although it is also possible to deploy the business rules on the same platform as the user interface in a three-tier application architecture, it is not recommended because of the software management problems which occur with many or dispersed user workstations are used.

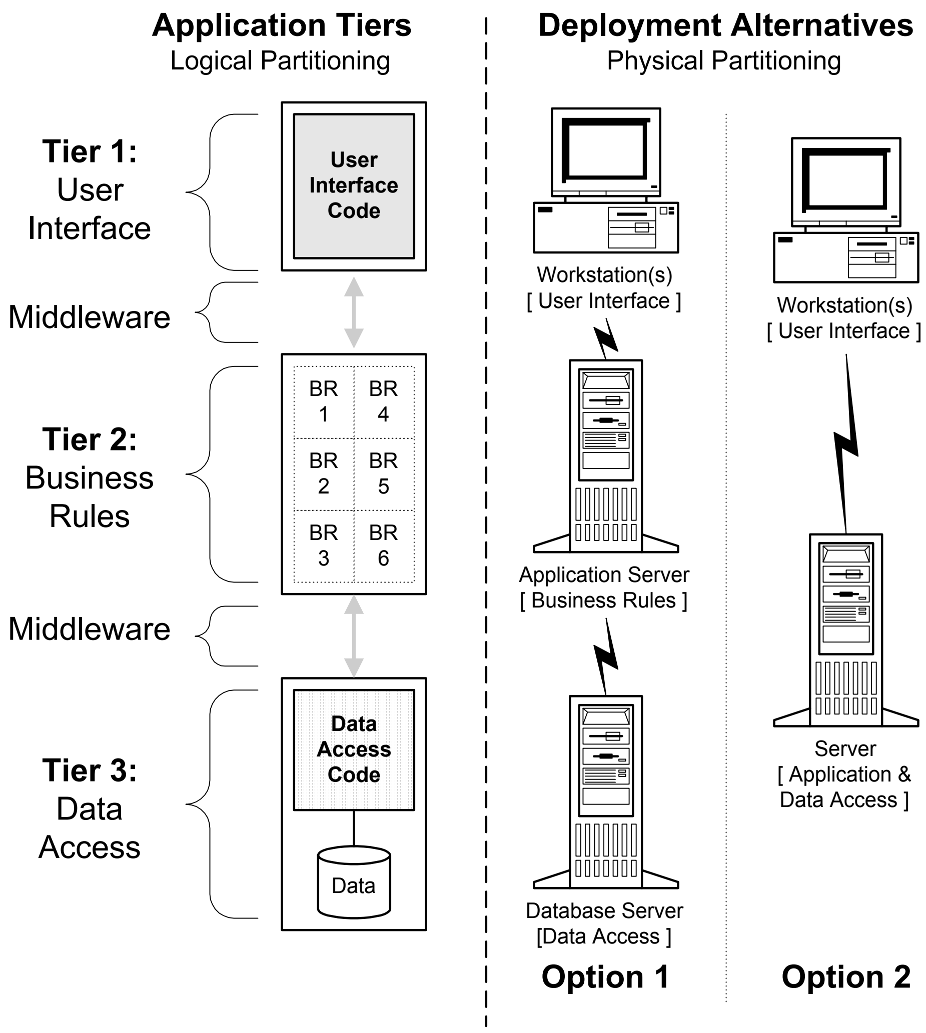

Figure 6. A three-tier client/server application

N-tier client/server applications

Many of the problems inherent in the state's existing application architecture of monolithic and two-tier applications can be overcome by implementing applications in an *N*-tier architecture. In an *N*-tier architecture, applications are partitioned into discrete units of functionality called "services". Each service implements a small set of related business rules or function points. "Business rules" support the processes the business follows. Business rules define what must be done and how it must be done.

Examples of business rules include:

- *Issue a check* IF (a) an invoice has been presented AND (b) the invoice is for work for which a purchase order was issued AND (c) the work has been performed AND (d) there is enough money in the bank to cover the check.

- *This student is eligible for early graduation* IF (a) she or he has completed the required work AND (b) she or he has achieved a grade point average of 3.0 AND (c) it is not yet time for her/him to graduate AND (d) she or he is at least 16 years old.

Business rules are processes followed when business events occur (i.e., business events are triggers for business rules). If business rules define what to do, business events define *why* it should be done. The following examples of business events might invoke the associated business rule:

- *A person applying for public assistance* triggers the business rules for "Determine eligibility for public assistance."
- *A person applying for a corporate charter* triggers the business rules for "Process application for incorporation."
- *A motorist driving erratically* triggers the business rules for "Traffic stop."
- *It's April 16* triggers the business rule for "Late tax return."

When a business rule must be modified to support changing business requirements, only the service that implements that business rule needs to be modified; the remainder of the application can remain intact. There is greater application adaptability for agencies. In the application illustrated in Figure 7, each business rule is implemented as a discrete executable (a "service") that can be requested by <u>any</u> client.

Since the business rules are implemented as separate executables, any combination of business rules may run on any combination of platforms. There is flexibility in selecting the platforms where application components can be deployed. As transaction loads, response time, and throughput change, any individual service can be moved from the platform on which it executes to another, more powerful platform. Application deployment is flexible and scaleable to accommodate greater transaction volumes.

Since business rules are implemented discretely, instead of tightly integrated with the graphical user interface, changes to business rules do not always require updates of code on the workstations accessing the application. It is easier to manage the deployed application.

Since business rules are implemented in discrete services, the same business rule can be invoked from users accessing the application from a GUI, from character terminals, or from web browsers. They can also be accessed by telephone from VRUs or by batch jobs. A separate interface tier provides programmer productivity and consistency of application behavior.

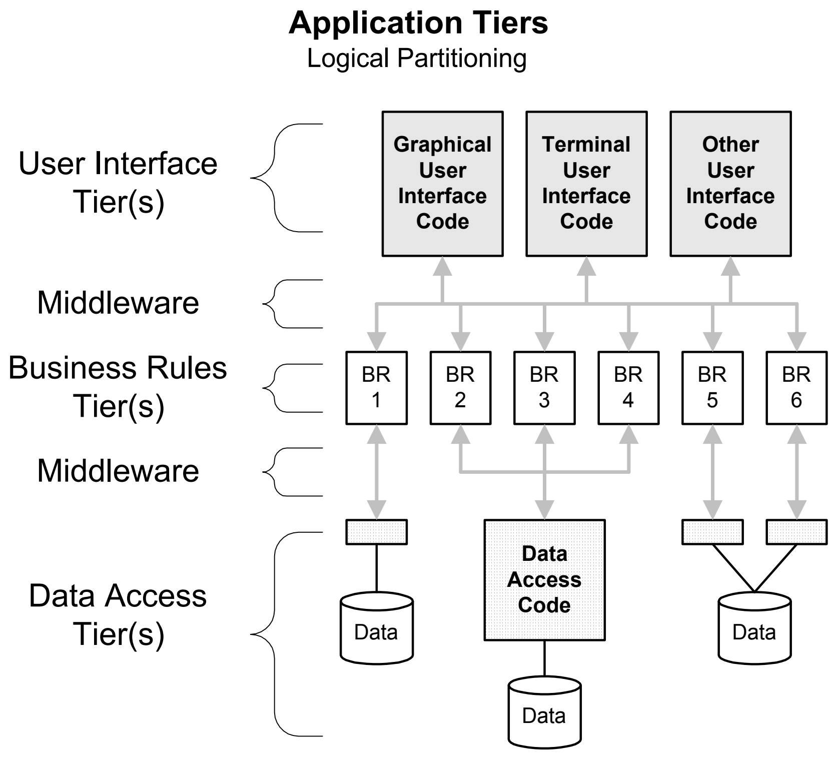

Figure 7. An N-tier client/server application

N-tier applications have the following advantages:

1. It is <u>easy to modify</u> them to support changes in business rules.
2. There is <u>less risk</u> modifying the code that implements any given business rule.
3. *N*-tier applications are <u>highly scaleable.</u>
4. An *N*-tier architecture offers the <u>best performance</u> of any client/server application architecture.
5. They can <u>support any combination of user interfaces</u>: character, graphical, web browser, telephones, and others.
6. They offer the highest potential for <u>code reuse and sharing</u>.

Service-Oriented Application Architecture

The maximum benefits of an *N*-tier architecture are realized when many *N*-tier applications are deployed across the state, sharing common software services that are accessible from any user interface. This is called a "service-oriented architecture". In this environment, any application can access any service, provided the application has the proper security permissions. In a service-oriented application architecture:

- Some services are shared by applications from multiple agencies.
- Others are shared by applications within a single agency.
- The rest are used by a single application.

The greatest strength of a service-oriented architecture is the potential for repeatable rapid development of new applications. Figure 8 illustrates *N*-tier applications in a service-oriented architecture.

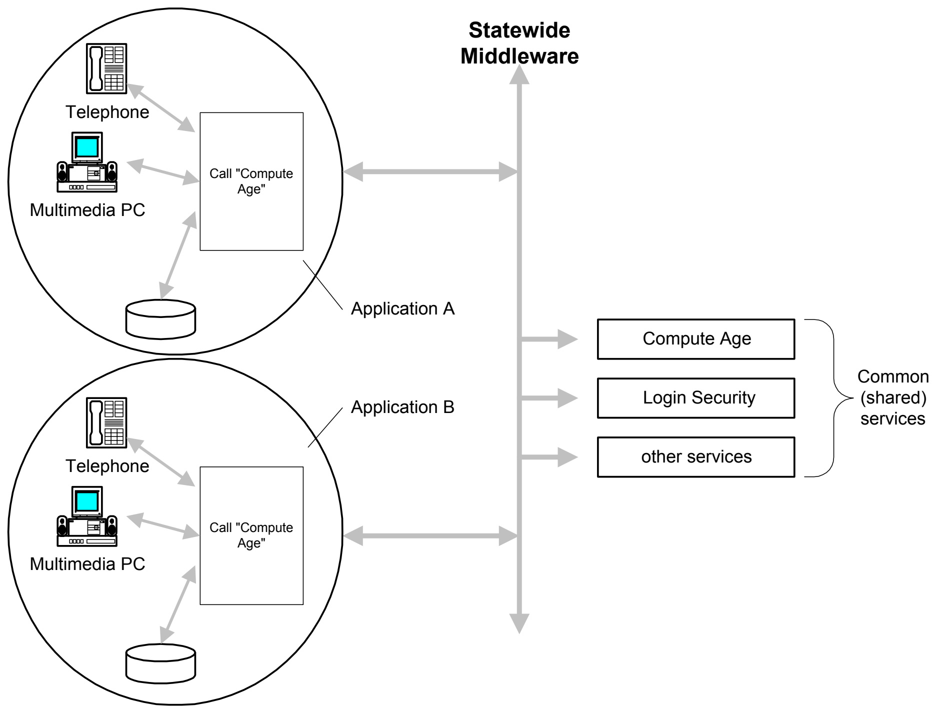

Figure 8. N-tier applications in a service-oriented architecture.

Application Technology Components

The technology components of applications are discussed below.

Application

Applications are the software that automates business processes. Regardless of what they do and the technology with which they are implemented, all applications have three general areas, known as "tiers", of functionality:

- ***Business rules***. Automates business processes using computer applications. As the business needs of agencies change, the business rules in the applications that support the agencies must be changed.
- ***Data access***. Automates the storage, search for, and retrieval of data by computer applications. In *N*-tier applications, changes in business rules do not usually require changes to the code that accesses data, but occasionally, they do.
- ***Interface***. Allows applications to communicate with other applications and with people. In *N*-tier applications, changes in business rules do not usually require changes in interface code, but sometimes, interfaces need to be updated when associated business rules have not changed (e.g., when changes occur in another computer system that interfaces with an application, or when users need a graphical user interface instead of a character-based interface).

Since applications interface with people, the *user interface* receives the most attention, but other interfaces are also important. Traditionally, people interfaced with computer applications using character terminals (e.g., 3270) or graphical user interfaces (e.g., Microsoft Windows). Recently new interfaces such as telephones (via VRUs), web browsers, and wireless devices have been introduced.

Middleware

Middleware is software that supports communications between the functional tiers of an application, between two or more different applications, and between applications and shared services. The role of middleware is to insulate application developers from having to understand the complexities of the computing environment and prevent them from having to hard code application interfaces. Refer to the Middleware Architecture chapter for more information.

Components

Components are program modules supporting a single business function or rule. Components shared by multiple applications must be designed for portability across platforms. Components used only within an application system can be developed in any supported language, with any development tool appropriate for the particular tier where they will be deployed. Refer to the Componentware Architecture chapter for more information about components.

Application Development Tools

The Application Architecture is independent of any specific technology. Its components, the interface, business rules, and data access code, can be implemented with *any development tool* in *any language* on *any platform* supporting the business needs of the application. Regardless of the tools selected, it is important that each tier be designed to be portable across platforms.

There are three approaches for selecting tools to develop client/server applications:

1. ***Best of breed***. Separate, specialized tools are used for each tier of an application. Middleware must be used to support communications between the different tiers.
2. ***Front end/back end***. Two different tools are used: a specialized user interface development tool and an integrated tool set that also provides middleware for the business rule and data access tiers. Middleware must be used to support communications between the user interface and other two tiers.
3. ***Integrated***. Integrated tool sets, or CASE tools, are used that generate code for all tiers of the application. These tools provide the middleware necessary to support communications between all tiers of the application.

Table 1 gives examples of tools used with each of these development approaches.

Tier/ Approach	Best of breed	Front-end/ Back-end	Integrated
User interface	• Centura: SQLWindows • Microsoft: Visual Basic • Sybase: PowerBuilder	• Centura: SQLWindows • Microsoft: Visual Basic • Sybase: PowerBuilder	• Antares: HURON • Bachman: Ellipse • Dynasty: Dynasty • Forte: Forte • Intersolv: APS • Seer: HPS • Texas Instruments: IEF
Data access	• Informix • Oracle • Sybase	• Magna Software: Magna X • Open Environment: Encompass	
Business rules	• COBOL • C • Traditional program generator		

Table 1. Development tools for various approaches

With the *N*-tier and service-oriented application architectures, two additional types of tools are required:

- Repositories, or libraries, to keep track of business rules that have been automated by components (see the Componentware Architecture chapter)
- Software management tools that provide version control, configuration management, and software distribution services (see the Systems Management chapter).

There is no "one size fits all" tool set that addresses the needs of all applications or that can be implemented on a statewide basis. The infrastructure of the Statewide Technical Architecture provides flexibility and choices for application development. Therefore, the selection of application development tools and intra-application middleware products is up to individual agencies -- as long as they support external calls to the state's middleware broker for inter-application communications and access to shared services (see the Middleware Architecture chapter).

Designing Manageable Applications

The state depends on the computer applications that support its business. If the application that supports an area of business were to go down or become unavailable, the state would be unable to deliver the services mandated by a program. For example:

- If the criminal history application becomes unavailable, law enforcement officers will be unable to determine during a traffic stop whether a driver should be considered dangerous.
- If the welfare eligibility information system becomes unavailable, the state will not be able to identify citizens needing public assistance.
- If the drivers license application becomes unavailable, the state would be unable to issue licenses to new drivers.

Due to the state's dependency on computer applications, applications must be managed as carefully as any other business-support infrastructure. *Application management is a necessity, not an option.* Application management requirements are as important to the enterprise as an application's functional requirements. Therefore, management requirements for an application should be documented during the requirements phase of the project.

Managing distributed applications (i.e., client/server applications) is more difficult than managing monolithic applications, because there are more pieces involved. The application itself has more components, and it has dependencies on more infrastructure components (e.g., networks, servers, workstations, software components, and databases). A client/server application may become unavailable if any component of the application fails or if any resource the application depends on is unavailable.

A client/server application will fail, for example:

- If one of its software components (e.g., a module that performs business rule processing) terminates abnormally.
- If the network connection between a client process and a server process becomes unavailable.

- If a database table the application must update is full.
- If a shared software service (e.g., validate social security number) cannot be accessed.

The application must be managed as a whole. This requires managing each component of the application and the infrastructure components it depends on. The ability to detect the causes of application failures, like those highlighted above, allows operations staff to respond quickly to restore service and to minimize the impact an application outage has on the state's business.

This section deals with designing applications to facilitate their management in production. This includes starting and stopping applications and their components; reconfiguring components; monitoring availability, errors, and performance; and controlling running applications.

Most management tools for the distributed environment deal with the infrastructure that distributed applications require -- machines, networks, routers, databases -- but not with the applications themselves.

Network and system management (NSM) tools can be used to manage applications as well as infrastructure components, but the applications must be instrumented to facilitate management. Instrumentation is source code that is added to each component of the application to facilitate its management. Instrumentation allows applications to provide feedback to the NSM tools and respond to commands issued by system administrators using the NSM tools.

When instrumenting applications, the costs of application management must be balanced with the business need for the application. Application management has costs associated in design, development, maintenance, and monitoring. Managing applications consumes system resources and impacts the performance of the application itself and other applications that share the same resources (e.g., platforms and networks). *The cost of application management is an investment that supports the state's ability to continue to do business.*

The management functions of applications should be compatible with NSM tools deployed to manage applications. SIPS is currently implementing a Simple Network Management Protocol (SNMP) compliant tool as the statewide NSM tool. To help agency developers instrument applications correctly, SIPS will specify the types of information applications must provide to be managed, as well as the types of commands to which applications must respond. All distributed applications -- client/server, object-oriented, and web applications -- must support the state's application management requirements.

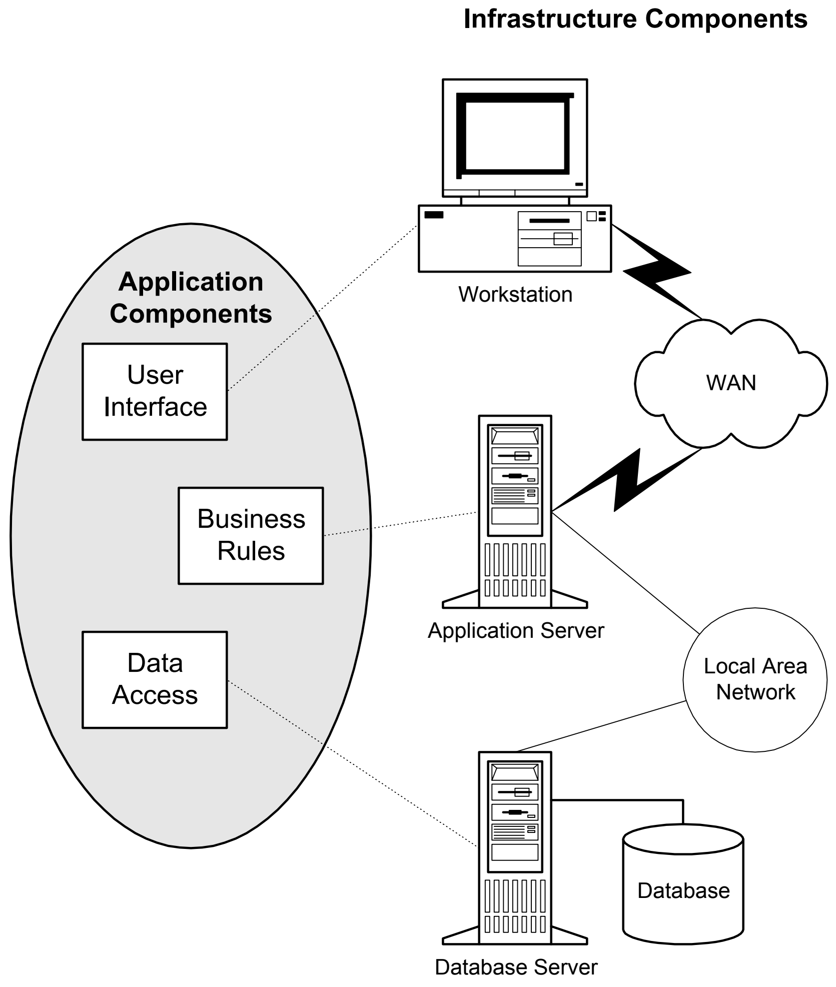

Figure 9. A distributed application in its operational environment.

Manageable Application Technology Components

Application management requires the following technology components:

Instrumented applications

Instrumentation in applications is code that provides information to or processes commands from the management environment. Applications, and each component within an application, should have the capability to:

- Report:
    - Events (e.g., "I just serviced a request")
    - Performance statistics (e.g., "it took .3 seconds to service a request")
    - Errors (e.g., "I was unable to service a request because it contained an invalid parameter")
    - Conditions (e.g., "there are 5 more requests in the queue")
- Receive and process commands, such as:

- Shut down the component
- Re-process the configuration file
- Begin using a different database.

Consistency in application instrumentation is easiest when all development teams use the same code templates for management services. Templates should include the basic instrumentation, and places for application teams to add the functional code required to support the application's mission (business rules).

```
/*** code template for servers ***/
/*** includes management functions ***/
Service_Name()
{
        call startup();
        call configure_service();

        while (get_request() not equal to 0 )
        {
                /*****************************
                /* business rule processing goes here
                /*
                /*
                /*****************************/
                send_reply();
        }
        call shutdown();
}

startup()
{
~~~ code for starting service ~~~
}

configure_service()
{
~~~ code for configuring service ~~~
}

trace_on()
{
~~~ code to turn tracing on ~~~
}

trace_off()
{
~~~ code to turn tracing off ~~~
}

shutdown()
{
~~~ code to gracefully exit ~~~
}

/***** end module ******/
```

Figure 10. A code template for services should include application management routines.

Agents

Agents are programs that collect status from applications, filter it, and report it to the system management framework. Please refer to the Systems Management Architecture for more information regarding agents.

Network and System Management (NSM) framework

The NSM framework monitors networks, systems, and applications by identifying alarms forwarded by agents. It alerts operations staff to the condition, so they can take corrective action. Please refer to the Systems Management Architecture for more information regarding NSM tools.

Management Information Base (MIB)

A MIB is a database that contains information about the application being managed. It includes dependencies on systems, other applications, databases, software components, etc. Application developers must supply some of the information for the MIB. Please refer to the Systems Management Architecture for more information regarding MIBs.

Management Protocols

NSM tools, agents, and the applications they manage must communicate with each other in a common language, or protocol. Please refer to the Systems Management Architecture for more information regarding management protocols.

Development Tools

Some application development tools support only certain application management protocols. If an application development team chooses to use a tool that includes support for an NSM tool, it should support the NSM tool being used statewide. Tools that support application management will make the developers job easier, and will help ensure that the application can be managed by the state's NSM tool.

ComponentWare

Componentware Architecture enables efficient reuse of existing application assets, faster deployment of new applications, and improved responsiveness to changing business needs. Reusable software components are the building blocks that make a system able to respond quickly to change.

As described earlier, traditional application programming techniques employed within the State of North Carolina resulted in an inventory of monolithic applications. Monolithic applications perform comprehensive business functions and operate independently from other applications. Making changes to a monolithic system is a major undertaking because changes in one area often cause problems in other areas.

Monolithic applications often incorporate the use of function calls and subroutines in an effort to divide the application into smaller, more manageable parts. This type of application is called a "modular application." Modular applications are easier to maintain than non-modular, but they still remain tightly integrated within a single application system. Though state-of-the-art at the time, modular development techniques have evolved into other techniques. These other techniques greatly enhance the efficiency of the application development cycle and minimize redundancy by incorporating shareable and reusable components. Sharing and reuse of components are the prime objectives of the Componentware Architecture.

Opportunities to share and reuse components exist today. There are many business functions that are common from agency to agency and are replicated in the applications that support them. For example, citizens requesting state services are usually asked a set of common questions: What is your name? What is your address? What is your phone number? What is your date of birth? etc. Once the data is collected, computer applications process the information. Across the state, these same functions are performed over and over in many different applications.

Figure 5-1 illustrates two application programs containing the code to perform the same function, "Compute Age". If there is a change to the "Compute Age" function, it must be located and changed in every application. For example, if dates are stored in the format YY/MM/DD and need to be changed to YYYY/MM/DD, then every program with the "Compute Age" function must be located, updated, relinked, recompiled, and retested.

Alternatively, if "Compute Age" were isolated as a common, reusable component, the impact of this change would be significantly less. When reusable and shareable components are used, individual applications contain application-specific logic, but call a common "Compute Age" component. If the date requirements change from YY/MM/DD to YYYY/MM/DD, the common "Compute Age" component is changed, tested and replaced. (See Figure 11) There is no need to search each application for a "Compute Age" routine. Identifying common, reusable components provides significant benefits for maintaining application code.

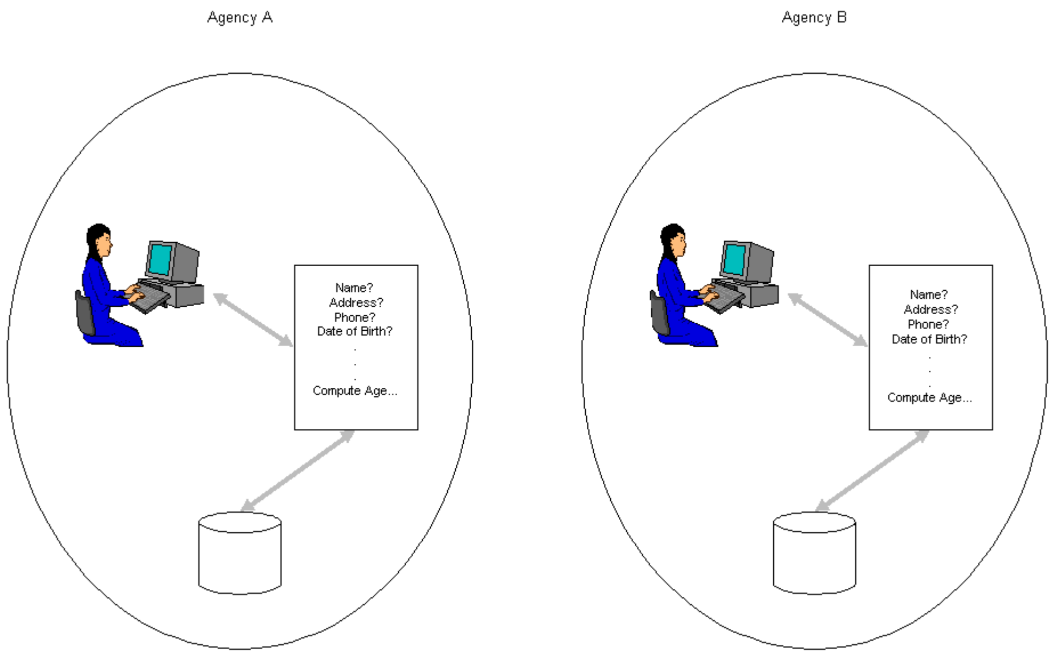

Figure 11. No sharing of common code between applications.

The benefits of sharing and reusing common components include:

- Productivity gain for the developers. Pre-built and pre-tested reusable components, ready for assembly by a developer, reduce the effort and time required to develop new or maintain existing applications.
- Consistency and accuracy of processing. Consistency and accuracy are achieved by having only one component responsible for a particular function. By eliminating the duplication of processes, the possibility of a process being performed more than one way is eliminated, thus reducing the potential for errors.
- Simplified testing. Once a component has been thoroughly tested, there is no need for further testing of that component when it is integrated with other components to form an application.

To achieve the benefits of sharing and reusing components, a successful reuse strategy must include:

- A reuse methodology consistently applied by application developers
- A component review board whose function enables the reuse program by reviewing projects and assisting with component reuse.
- A technique for identifying reusable components, also known as "harvesting." Harvesting involves the examination of legacy applications for

the purpose of identifying functions that can be isolated into standalone program modules (i.e., components).

- A “wrappering” or “encapsulation” technique. Legacy applications can be formed into components by using a technology called “wrappering” or “encapsulation”. Code is implemented that “wraps” an API around a legacy service. A wrapper is used with legacy applications for the purpose of implementing reusable components.
- Documentation for each component that includes a well-defined set of input and output parameters for each interface option provided.
- A library, or repository, of information about reusable components.

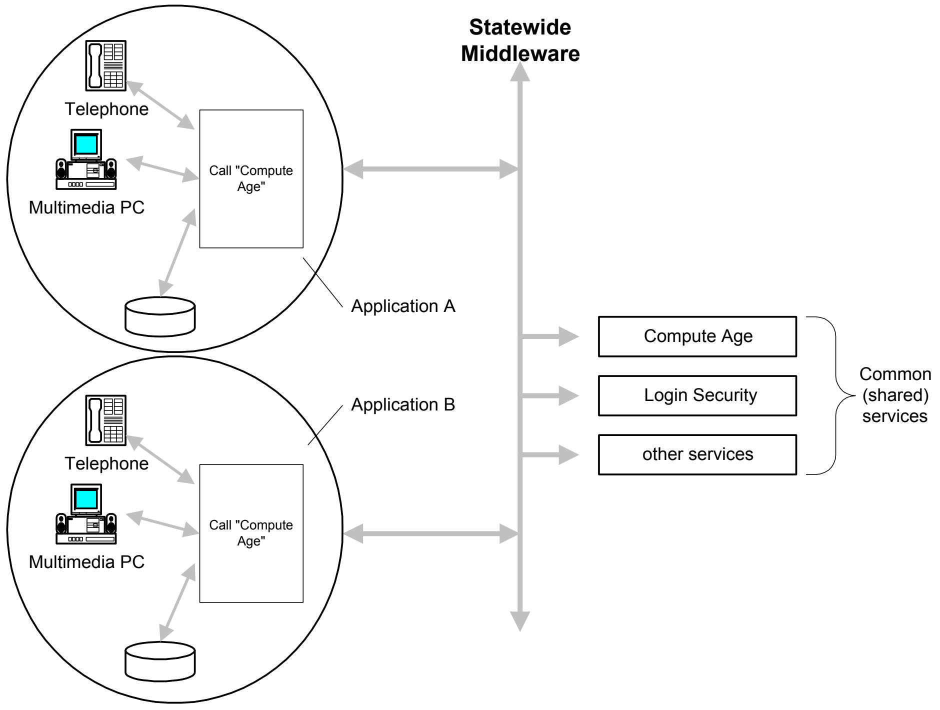

Figure 12. Sharing a common component.

Integrated error and exception handling capabilities that enables each component to operate independently from other components and applications.

Reuse Methodology

A successful implementation of an *N*-tier, reusable component service-oriented architecture is not solely dependent on the ability to develop reusable components.

Success also depends on the ability to provide the tools and management of the components for reuse.

As discussed in the Data Architecture chapter, the state benefits by having a federated data model where data is defined consistently and shared. Federated data is data available for use within a single agency, between multiple governmental organizations, and across the state. Application code maintaining federated data should also be reused and shared. If more than one program updates a piece of data, then there is the risk of one program performing the update slightly different from another program. If multiple programs or applications update the same data, there is a risk to the integrity of that data.

The development of a Componentware Architecture would best be implemented incrementally over time through ongoing projects. The key is to put in place a solid strategy. If a Componentware Architecture is not explicitly designed and actively managed, the result will be a more difficult development environment than is currently in place in the state. The key elements of a reuse program are:

- Inventory.
- Catalog.
- Reuse administrator and facilitator.
- Methodology.
- Design standards and principles.
- Measurement.
- Quality assurance.
- Performance incentives.

A reuse program should be established to enable the reuse methodology statewide. A reuse methodology should be incorporated into the system development life cycle.

Componentware Architecture recommends the forming of a component review board that reviews projects and assists with the harvesting and implementation of components. The component review board should be comprised of key business users from across the enterprise. The focus of the review board is to enable the reuse program. In order to ensure a successful reuse program, the right people have to be involved with the knowledge and the authority to negotiate the definitions of reusable components.

Techniques for Reusing Components

The notion of reuse is not new. Application developers have been reusing code for many years. The Componentware Architecture builds upon familiar paradigms. Examples of code reuse are:

Including code from one program into another. There are two variations for including code: (1) copying the source code from one program directly into the source code of another; or (2) using "include" files or copybooks. The drawback with these methods of reuse is the expense of adapting to and maintaining changes. If a piece of copied code implements the logic necessary to carry out a particular business rule, then every time a business rule changes the change has to be replicated everywhere that code exists. After the change is made, each program must then be retested. Even if the rule is implemented using an "include" file or copybook, all programs containing that "include" file or copybook must be recompiled and retested.

1. **Linking programs with libraries of compiled program object modules**. Linking programs is accomplished 1 of 2 ways: using the link facility provided with the computer operating system at compile time; or by using Dynamic Link Libraries (DLL) at runtime. This method of reuse is better than copying the code from one program to another because the business logic only exists once. However, if a module implementing the business rule changes then any programs linked with that module need to be identified, relinked, and retested.
2. **Calling a service that performs the desired task**. Programming function libraries (and operating system service libraries) employ this method of reuse. For example, most programming languages supply mathematical function libraries so programmers do not have to write mathematical functions (e.g., a square root function). Similarly, operating systems supply calls for system service functions (e.g., file management). The use of common components is an extension of this concept for business functions.

Calling a service that performs the desired task is preferred method of code reuse and is the recommended technique for using components in the Componentware Architecture. This method supports the *N*-tiered design recommended by the Application Architecture.

Types of Services provided by Components

Reusable components can classified into different types, as illustrated in Figure 3:

- *Application services.* These components perform business rules (e.g., "Compute Age"), provide access to the business data (e.g., "Get Citizen Record"), or communicate with other systems using application program interfaces (e.g., "Reorder Books"). Application services that handle data should be the only method by with data is accessed.

- *User interface services.* User interfaces facilitate interaction between people and application systems. Occasionally, changes in business rules require changes in interface code. More often, new interfaces need to be added to an application system. For example, a voice response unit (VRU) or web browser may be added to an application originally built with 3270 terminal interface. User interface services enable the applications to be accessed by any type of user interface. The objective is to design the system in such a manner that it won't matter what user interface is being used to access an application, because the flow of information will still be the same. Typical user interfaces include:
    - Graphical user interfaces (i.e., GUI).
    - Non-graphics terminals (e.g., UNIX terminals or 3270 terminals).
    - Web browsers.
    - Point of sale devices (e.g., cash registers).
    - Telephone interfaces via VRUs.
- *Support services.* Services that typically provide operating system type functions, such as printing, faxing, and imaging, that are typically provided by purchased packages. If purchased they should easily integrate into an *N*-tier environment.
- *Core services.* These components provide basic application infrastructure services such as security, naming, and directory services.

The importance of the classification of components into application, support, core, and user interface services is that each has their own special benefits. Support and core services are typically purchased packages integrated with the rest of the operating system. These services should not be a concern for application developers; instead, they should exist and be used as integral components of the system architecture just as a keyboard is an integral component of a PC. The goal is to focus the efforts of the application developers on creating new business application services that can be shared with other applications in the state.

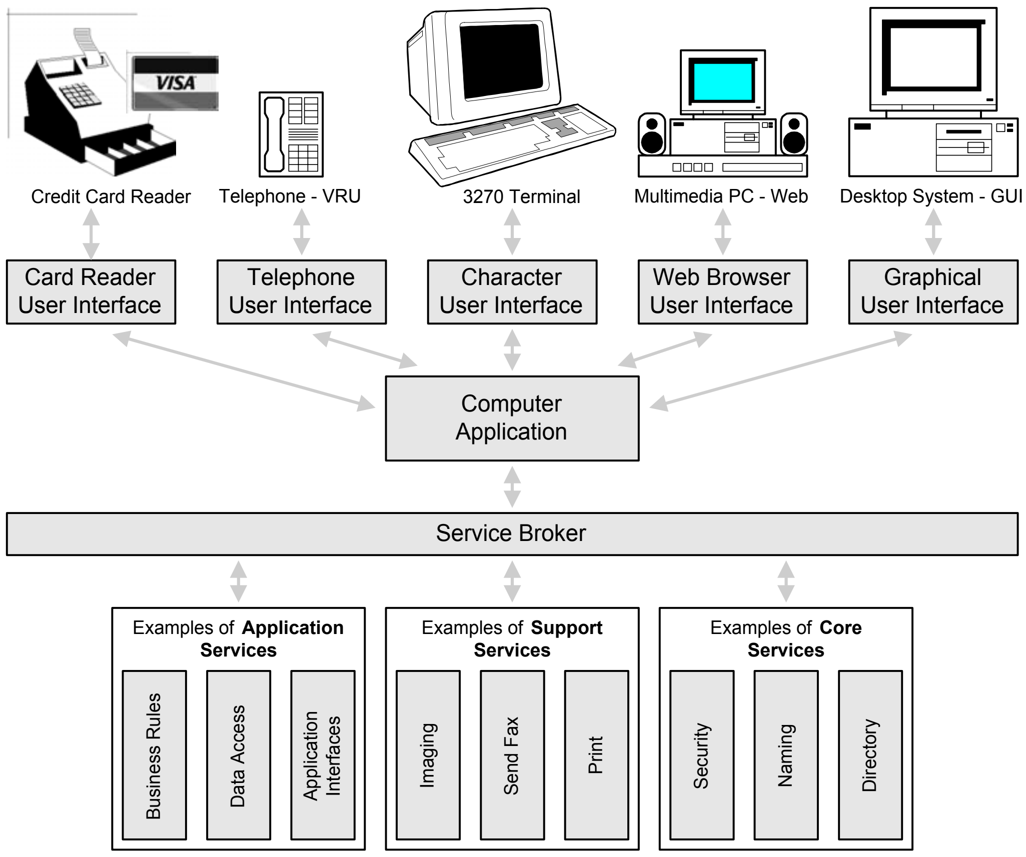

Figure 13. Various types of components.

Object-oriented Components

Object-oriented components encapsulate both the business logic and the data accessed by the business logic. They have the potential to become intelligent, self-managing entities, allowing for more simplified management.

Properly wrapped, object-oriented components can be accessed by both object-oriented and traditional applications. Likewise, non-object oriented components can be accessed by both object-oriented and non-object oriented applications.

There are several reasons why objects are discussed here as "potential", but not included in the current migration strategy:

1. Object technology radically alters the way software is developed. Many of the same benefits can be achieved using traditional technology familiar to development staff with legacy skills. Non-object-oriented components can:

   - Plug-and-play across the network.
   - Run on different platforms.

- Coexist with legacy applications and object-oriented applications.
- Protect data without encapsulation by providing the only means of accessing the data.

2. Objects are still an emerging technology. Standards and techniques are still evolving. There are currently two different object standards:

   - The Common Object Request Broker Architecture (CORBA). This is the model promoted by the Object Management Group (OMG). As a consortium of corporations, the OMG was founded specifically to create distributed object industry standards. The membership list is over 700 members.

     Components developed with the CORBA standard are language neutral so components written in one language can interoperate with components in other languages. Since CORBA supports inheritance, components developed with this standard are very reusable and can be deployed on virtually every computing platform in the industry.

   - The Object Linking and Embedding/Distributed Component Object Model (OLE/DCOM). These objects are based on a defacto standard developed by Microsoft Corporation. OLE/DCOM specifies intra-application and inter-application communication interfaces between OLE-compliant components. OLE has been an accepted defacto standard for objects close to the end-user for some time.

     DCOM has drawbacks, though. First, the DCOM standard supports encapsulation but does not support inheritance. Also, objects developed using the OLE/DCOM are limited to the Windows™ family of operating systems while CORBA objects offer broader platform support. CORBA objects can be wrapped with code that will allow them to interface with OLE/DCOM objects.

Since most of the gains of object technology can be obtained with a service-oriented architecture, the migration to object-oriented programming should be delayed by state application developers until the marketplace matures, the necessary skills have been acquired, and a clear technical standard emerges. At the same time, agencies should not reject purchasing applications developed by organizations experienced in object-oriented technology, since a vendor's object model will not negatively impact the state.

ComponentWare Technology Components

The statewide Componentware Architecture requires the following technology components to be successful.

Components

Components are program modules providing a complete package of business functionality. Shared components must be designed for portability across platforms.

Components within an application system can be developed in any supported language, with any development tool appropriate for the particular tier where they are deployed.

Application Program Interface (API)

Each component developed to be reused must have a well-documented Application Programming Interface (API). The API defines:

- The parameters that must be passed to the component, including required and optional parameters.
- The output to be returned and its output.

For example, if a "Compute Age" component computes an age (in years) based on a birth date, its API might require a birth date, passed in a string, in MM/DD/YYYY format. The "Compute Age" component then returns an integer in the range of 0 - 150.

The API must be available to programmers who use a component.

Interface Definition Language (IDL)

Application programs must be able to pass parameters to and receive results from components. To accomplish this, application developers must understand the API of the components.

Interface Definition Language (IDL) provides a means for component developers to describe a component's API. The IDL source code is compiled and made available to programmers needing to use the component. By including in an application the compiled IDL description of a component's API, the application can interact successfully with the component.

Note that a component can be changed without requiring modifications to the applications that use it, as long as the component API does not change. For example, if the "Compute Age" component is changed to accelerate the age calculation function, the programs calling "Compute Age" do not have to be modified. "Compute Age" still receives a string containing a date in MM/DD/YYYY format and will return an integer in the range of 0 - 150.

Repository

A repository is a library assisting programmers in finding components that can be used to construct applications. It contains information about the components available to be reused in the application development process. For example, the repository might contain an English language description of what a component does, the API required to use it, and the IDL description of the API.

There are two choices for developing a component repository. It can be purchased from a vendor and configured as a reusable component repository, or it can be custom developed.

Harvesting legacy applications for candidate components can initially populate a component repository. Legacy systems are a good place to start for populating the component repository because they contain all the current business rules. The problem with building components from legacy systems is that most legacy applications are not designed for component use and may require major modifications for component extraction.

Accessibility

Accessibility Architecture provides standards for accessing information by persons who must receive it in a form that is different from the manner in which it is normally presented. The Chapter provides methods of designing systems so that the largest number of people can utilize the information contained therein. Examples include people with sight, hearing, mobility, or cognitive impairments. The subject matter also addresses low-bandwidth network connections; along with legacy, intermediary and "thin" clients, which have limited display or functional capabilities.

The purpose of this Chapter is to provide a set of recommended guidelines for State developers, designers, procurement officers and commercial suppliers of electronic and information technology and services that will result in access to and use of the technology and information by all individuals, especially those with disabilities. This Chapter represents minimally acceptable standards. All entities involved in the design, production, and procurement process of relevant electronic and information technology are strongly encouraged to go beyond these standards to maximize the accessibility and usability of products by all individuals.

Electronic and Information technology (E&IT) used by the State government shall be accessible to and usable by all individuals, including those with disabilities. Being accessible and usable by people includes being able to perform all the regular operating functions of the E&IT including input and control functions, operation of any mechanical mechanisms, and access to information displayed in visual and auditory form. It also includes the ability to work with the assistive technologies used to access E&IT and should not interfere with the assistive technologies used on a daily basis by people with disabilities. Documentation and services associated with E&IT shall also be accessible and usable.

These architectural guidelines apply to a full range of E&IT including those used for communication, duplication, computing, storage, presentation, control, transport and production.

Usability of information technology becomes a serious issue for Americans with disabilities when they are excluded from the e-commerce and e-government due to the inaccessible design of information technology. Those who "can" and "cannot" are finding a growing number of access barriers, such as inaccessible web sites, software incompatibility with adaptive devices, and voice automated systems inaccessible to adaptive telephones.

Fortunately, a closer look at the design features of accessible technology reveal that the benefits extend beyond the community of people with disabilities. For example, accessible web design enables the very functionality needed for dynamic, web-based transactions - whether or not it is for business transactions, voting or long-distance learning. In addition, CD and videotapes can be archived through captioning and electronic textbooks can be made accessible. Even illiterate populations can access the web by listening to screen readers audibly reading the text on the web page.

But perhaps the most significant benefit for the global economy is the fact that accessible design enables low technology to access high technology, thereby contributing to a stable, sustainable electronic infrastructure. People with slow modems and low bandwidth can access the electronic content of the web even if they do not have the state-of-the-art computer equipment. Likewise, people with personal digital assistants and cell phones can access the content of systems incorporating accessible design features.

It is important to remember that accessibility is a quality, not a quantity. It is not finitely measurable. Thus, if a solution meets the principles, but does not utilize the standard, than it may still be accessible. The Technical Topics section outlines recommended standards for identifying whether E&IT provides comparable access. However - The principles represent the actual measure of accessibility. This is necessary in order to ensure that the requirements of accessibility are met fully, effectively, and efficiently--both now and in the future as the technological capability of E&IT evolves. Accessibility provisions should not only permit basic access, they should also allow people with disabilities to maximize the use of the abilities they have.

Goals

The Accessibility Architecture chapter documents the approach for the state to provide the maximum achievable access for its information. The goals of the Accessibility Architecture are to:

- Ensure that all new Electronic and Information Technology produced, procured, or developed by agencies subject to IRMC policy will be accessible.
- Provide a cross-organizational structure of interoperability with technologies that increase access.

- Provide access to data in formats that separate presentation from content.
- Provide guidelines for accessible training and support.
- Utilize national and international standards to achieve these goals.
- The Accessibility Architecture chapter consists of the following technical topics:
- Software Applications
- Hardware Accessibility - To be added later
- Training - To be added later
- Support - To be added later

Definitions

Electronic and Information Technology

The term "Electronic and Information Technology" (IT) shall mean Electronic and Information technology used by agencies under the jurisdiction of the IRMC that is used in carrying out Electronic and Information activities, involving any form of Electronic and Information, where:
"Electronic and Information Technology" includes but is not limited to equipment, hardware, computers, software, firmware and similar procedures, systems, ancillary technologies, technologies which cause content to be active in any way, documentation, services (including support services) and related resources. It includes subsystems, interconnections, and interconnected systems.

Electronic and Information Activities

"Electronic and Information Activities" include, but are not limited to, the creation, translation, duplication, serving, acquisition, manipulation, storage, management, movement, control, display, switching, interchange, transmission, or reception of data or Information. Electronic and Information activities include delayed presentation activities such as Electronic and Information servers and messaging systems as well as synchronous, real-time communication activities.

Electronic and Information

Electronic and Information includes, but is not limited to, voice, graphics, text, dynamic content, and data structures of all types whether they are in electronic, visual, auditory, optical or any other form.

Undue burden

Undue Burden means significant difficulty or expense. In determining whether an action would impose an undue burden on the operation of the agency in question, factors to be considered include:

- The nature and cost of the action needed to comply with this section;

- The overall size of the agency's program and resources, including the number of employees, number and type of facilities, and the size of the agency's budget;
- The type of the agency's operation, including the composition and structure of the agency's work force; and
- The impact of such action upon the resources and operation of the agency.

Accessibility

The capacity of an Electronic and Information Technology to enable any user to engage in Electronic and Information Activities.

Accessible

Electronic and Information Technology that enables any user to engage in Electronic and Information Activities.

Accessibility Technology Components

Software Application components are divided into three sections: Internally developed, Externally developed, and Internet (Web-based) applications and services.

The following technology components are required for accessibility in software application components.

Internet (Web-based) Applications and Services

Usability of information technology becomes a serious issue for Americans with disabilities when they are excluded from the e-commerce and e-government due to the inaccessible design of information technology. Those who "can" and "cannot" are finding a growing number of access barriers, such as inaccessible web sites, software incompatibility with adaptive devices, and voice automated systems inaccessible to adaptive telephones.

Fortunately, a closer look at the design features of accessible technology reveal that the benefits extend beyond the community of people with disabilities. For example, accessible web design enables the very functionality needed for dynamic, web-based transactions - whether or not it is for business transactions, voting or long-distance learning. Even illiterate populations can access the web by listening to screen readers audibly reading the text on the web page.

But perhaps the most significant benefit for the global economy is the fact that accessible web design enables low technology to access high technology; thereby contributing to a stable, sustainable electronic infrastructure. People with slow modems and low bandwidth can access the electronic content of the web even if they do not have the state-of-the-art computer equipment. Likewise, people with personal digital assistants and cell phones can access the content of web sites incorporating accessible web design features.

In October 1994, Tim Berners-Lee, inventor of the Web, founded the World Wide Web Consortium (W3C) at the Massachusetts Institute of Technology, Laboratory for Computer Science [MIT/LCS] in collaboration with CERN, where the Web originated, with support from DARPA and the European Commission. In April 1995, the INRIA (Institut National de Recherche en Informatique et Automatique) became the first European W3C host, followed by Keio University of Japan (Shonan Fujisawa Campus) in Asia in 1996. W3C continues to pursue an international audience through its Offices worldwide. The W3C writes the standard for languages of the Web, including HTML, XML, Xforms, SmiL, and XHTML, along with architectural and interoperability standards.

Utilization of the World Wide Web Consortium's (W3C) Web Content Accessibility Guidelines provides the best model to maximize usability. The recommended best practices in this section pertain to the implementation.

Internally developed components and programmatic elements

This technology component will be addressed in a future release of this document.

Externally developed components and programmatic elements (including COTS)

This technology component will be addressed in a future release of this document.